Programming Languages

Lecturer: William W.Y. Hsu
Chapter 7 – Data Types
Data Types

- Computers manipulate *sequences of bits*.
- We manipulate *higher level data* (numbers, strings, etc.)
- *Data types* transform bits into higher level data.
Data Types

- Types provide **implicit context**.
  - **Compilers can infer information**, so programmers write less code.
  - The expression `a+b` in Java may be adding two `integer`, two `floats` or two strings depending on **context**.

- Types provide a set of **semantically valid operation**.
  - Compilers can **detect semantic mistakes**.
  - Python’s `list` support `append()` and `pop()`, but complex numbers do not.
Type Systems

- A type system consists of:
  - A mechanism to define types and associate them with language constructs.
  - A set of rules for “type equivalence,” “type compatibility,” and “type inference.”
Type Systems: Type Checking

- **Type Checking** is the process of ensuring that a program obeys the language’s type compatibility rules.
  - Strongly typed.
  - Weakly typed.
Strongly Typed

- **Strongly typed** languages **always detect type errors**.
  - All **expressions and objects** must have a type.
    - This includes functions!
  - All operations must be applied in **appropriate type contexts**.
- **Statically typed** languages are **strongly typed** languages in which **all type checking occurs at compiled time**.
Weakly Typed

- In **weakly typed** languages “anything can go.”
  - Perl.
Examples

- Common Lisp is strongly typed, but not statically typed.
- Ada is statically typed.
- Pascal is almost statically typed.
- Java is strongly typed, with a non-trivial mix of things that can be checked statically and things that have to be checked dynamically.
Common terms

• Discrete types – countable
  ▫ integer
  ▫ boolean
  ▫ char
  ▫ enumeration
  ▫ subrange

• Scalar types - one-dimensional
  ▫ discrete
  ▫ real
Common terms

• Composite types:
  ▫ records (unions)
  ▫ arrays
  ▫ strings
  ▫ sets
  ▫ pointers
  ▫ lists
  ▫ files
Typing and “Degree of Abstraction”

Very strongly typed languages won’t allow “implicit conversion.”
What is a Type?

- Three points of view
  - **Denotational**: Set of values.
  - **Constructive**: A type is “built-in” or “composite”.
  - **Abstraction-based**: A type is an interface that defines a set of consistent operations.
Strongly Typed

• Under assembly, any register can take any type of value (e.g., integer, string).
• Under **strongly typed languages**, a variable can only take values of a particular type.
• For example, “int a” can only have values of type “integer”.
Weakly Typed

- Weakly typed languages infer meaning at run-time.
  - **Advantage**: Increase Speed of development.
  - **Disadvantage**: Less error checking at compile time.
- Not appropriate for low-level programming or large programs.
Denotation

- Under denotation, a value has a given type if it belongs to a set.
- An object has a type, if its value is guaranteed to be in a certain set.
- A set of values is called a domain (i.e., its type).
- Similar to enum in C.
Built-in Types

- Built-in/primitive/elementary types:
  - Mimic hardware units.
  - boolean, character, integer, real (float).
- Implementation varies across languages.
- Characters are traditionally one-byte quantitates using the ASCII character set.
Built-in Types: Unicode

- Newer languages have built-in characters that support Unicode character sets
- **Unicode is implemented using two-byte quantities.**
- This is very important for moving legacy code.
Built-in Types: Numeric Types

• Most languages support **integers and floats**.
  ▫ (Their value range is implementation dependent)
• Some languages support other numeric types
  ▫ Complex Numbers (e.g., Fortran, Python)
  ▫ Rational Number (e.g., scheme, common Lisp)
  ▫ Signed and Unsigned integers (e.g., C, Modula-2)
  ▫ Fixed point Numbers (e.g., Ada, Cobol)
• Some languages distinguish numeric types depending on their precision.
Composite

- A composite type is created by applying type constructors to simpler types.
  - Records
  - Structs
  - Arrays
  - Sets
  - Classes
Classification of Types: Enumerations

- **Enumerations** improve program readability and error checking.
- First introduced in Pascal (but also exist in C):
  - `type weekday = (sun, mon, tue, wed, thu, fri, sat);`
  - They are **defined in order**, so they can be used in enumeration controlled loops.
Classification of Types: Subranges

- **Subranges** define a *valid range of values* for a variable.
  - Type `test_score = 0..100;`
- The improve *readability* and *error checking*. 
Classification of Types: Orthogonality

• Recall, orthogonality means that all features behaves consistently.
  ▫ $a=b$ always denotes assignment.
  ▫ Pascal is more orthogonal than Fortran, (because it allows arrays of anything, for instance), but it does not permit variant records as arbitrary fields of other records (for instance).
• This makes life much easier when reasoning about different types.
Type Checking

- Now that we’ve discussed the basics of types, let’s go back to equivalence, compatibility, and inference.
Type Checking

- **Type Equivalence**: When are the types of two values are the same?
- **Type Compatibility**: Can a value of A be used when type B is expected?
- **Type Inference**: What is the type of an expression, given the type of the operands?
Type Equivalence

- Type Equivalence is defined in terms of *structural* and *name equivalence*.
  - Compatibility is the more useful concept, because it tells you what you can DO.
Structural Equivalence

- Two types are structurally equivalent if they have the same components put together in the same way.

```c
typedef struct{int a,b;} foo1
```

```c
typedef struct{int a,b;}
{foo2
```
Structural Equivalence

typedef struct{
    char *name;
    char *addre;
    int age;
} student;

... but probably not intentional.

typedef struct{
    char *name;
    char *addre;
    int age;
} school;
Name Equivalence

- Name equivalence assumes that two definitions with different names are not the same.
- Solves the “student-school” problem.
Name Equivalence: Aliases

- Under name equivalence it is possible to define a new type via:
  
  ```
  TYPE new_type = old_type;
  ```

- Such a construction is called an **alias**.

- Two ways to interpret an alias:
  - **Strict name equivalence**: `New_type` is a different type than `old_type`.
  - **Loose name equivalence**: `New_type` is the same type as `old_type`. 
Problems

TYPE celsius_temp = REAL;
   farhen_temp = REAL;
VAR c: celsius_temp;
   f: farhen_temp;
...

f:=c;(* probably should be an error*)
Type Conversion

- A value of one type can be used in a context of another type using type conversion or type cast.
- Under a converting type cast, the underlying bits are changed:

```c
int i;
float f= 3.4;
i = (int) f;
/* runtime */
```
Type Conversion

- Under a non-converting type cast, the underlying bits are not altered.

```c
int i;
float f = 3.4;
i = *((int*) & f);
/* Compile time*/
```
Type Compatibility

- Most languages **do not require type equivalence in every context**.
- Two types **T and S are compatible** in Ada if any of the following conditions are true:
  - T and S are equivalent.
  - T is a subtype of S.
  - S is a subtype of T.
  - T and S are arrays with the same number elements and same type of elements.
Type Compatibility

- Type **coercion** allows a value of one type to be used in a context that expects another.

```c
short int s;
unsigned long int l;
...
s = l;
```

- This makes the system type weaker.
Type Compatibility

- Coercion rules are a relaxation of type checking.
  - Recent thought is that this is probably a bad idea.
  - Languages such as Modula-2 and Ada do not permit coercions.
  - C++, however, goes hog-wild with them.
  - They're one of the hardest parts of the language to understand.
- Make sure you understand the difference between:
  - Type conversions (explicit).
  - Type coercions (implicit).
  - Sometimes the word 'cast' is used for conversions (C is guilty here).
Generic Reference Types

- It is often useful to have a **generic reference type** that can hold any type of object:
  - In Java this is **Object**
  - In C and C++ this is **void** *

```c
void* v;
int* i;
...
v=i;
```
Type Inference

- Usually the type of the overall expression is easy.
- However, for subranges and composite objects is not so simple.
- In Pascal:

```pascal
type Atype = 0..20;
Btype = 10..20;

var a: Atype;
    b: Btype;

... 

a+b;
```
- What is the type of a+b?
Records

- **Records** (structs in C and C++) allow for a collection of related data to be manipulated together.

```c
struct foo{
    int a;
    int b;
}
```
Record: Memory Layout

- There may be **holes** in the allocation of memory.

```haskell
type ore = record
    name : two_char;
    atom_num: integer;
    atom_weight: real;
    met: Boolean;
end;
```

Holes waste space and complicate comparisons.
Other Arrangements

- **Packed layouts** require multiple instructions for accessing elements and assignments.
Variant Records

- A variant record (union) provides two or more alternative fields or collections of field but only one bit is valid at any given time.

```c
struct element{
    char* Full_name;
    union{
        int atom_num;
        char atom_sym[2];
    }
}
```

element can contain atom_num or atom_sym, but not both.
Variant Records

```c
struct element{
    char* Full_name;
    union{
        int atom_num;
        char atom_sym[2];
    }
}
```

![Diagram of variant records structure]
Arrays

• Arrays are usually stored in *contiguous* locations.

Row major order

Column major order
Arrays

char days[][10] = {
    “sun”, “mon”, “tue”,
    “wed”, “thu”, “fri”,
    “sat”
};
Arrays

char *days[] = {
    "sun", "mon", "tue",
    "wed", "thu", "fri",
    "sat"
};
Arrays

Arrays

S3 = size of element
S2 = (U2-L2+1) * S3
S1 = (U3-L3+1) * S2

Address of A[i,j,k] = Address of A + (i-L1)*S1 + (j-L2)*S2 + (k-L3)*S3

Optimized:
(i * S1) + (j * S2) + (k * S3) + address A -[(L1 * S1) + (L2 * S2) + (L3 * S3)].
The phrase [(L1 * S1) + (L2 * S2) + (L3 * S3)] can be determined at compile-time.
Strings

• Strings are really just arrays of characters.
• They are often special-cased, to give them flexibility (like polymorphism or dynamic sizing) that is not available for arrays in general.
• It's easier to provide these things for strings than for arrays in general because strings are one-dimensional and (more important) non-circular.
Sets

• We learned about a lot of possible implementations.
  ▫ Bitsets are what usually get built into programming languages.
  ▫ Things like intersection, union, membership, etc. can be implemented efficiently with bitwise logical instructions.
  ▫ Some languages place limits on the sizes of sets to make it easier for the implementer.
Files and Input/Output

- Input/output (I/O) facilities allow a program to communicate with the outside world.
  - Interactive I/O and I/O with files.
- Interactive I/O generally implies communication with human users or physical devices.
- Files generally refer to off-line storage implemented by the operating system.
- Files may be further categorized into
  - Temporary.
  - Persistent.
Heap-based Allocation

- The **heap** is a region of storage in which sub-blocks can be allocated and deallocated.
Issues with Heap Allocation

• Pointers
  ▫ Used in value model of variables.
  ▫ Not necessary for reference model.

• Dangling References

• Garbage Collection
Pointers

• Pointers serve two purposes:
  ▫ Efficient (and sometimes intuitive) access to elaborated objects (as in C)
  ▫ Dynamic creation of linked data structures, in conjunction with a heap storage manager.

• Several Languages (e.g., Pascal) restrict pointers to accessing things in the heap.

• Pointers are used with a value model of variables
  ▫ They aren’t needed with a reference model (already implicit).
C Pointers and Arrays

- C Pointers and array:

  int *a == int a[]
  int **a == int *a[]
C Pointers and Arrays

- But equivalencies don’t always hold:
  - Specifically, a deceleration allocates an array if it specifies a size for the first dimension.
  - Otherwise, it allocates a pointer.

```c
int **a, int *a[]; // Pointer to pointer to int
int *a[n];         // n-element array of row pointers
int a[n][m];       // 2-D array
```
Function Pointers

- Consider the following function declaration in C:
  ```c
  int add( int a, int b) // Function add
  {
    return a+b;
  }
  ```
- This is a function taking 2 int arguments and returning an int.
- add is the name of this function.
  - The type of this function is int (*a)(int, int);
#include <stdio.h>

int add( int a, int b) { return a+b; }
int sub( int a, int b) { return a-b; }

int main( void )
{
    int (*func)(int a, int b);

    func = add;
    printf( "%d\n", func( 1, 2 ) );
    func = sub;
    printf( "%d\n", (*func)( 1, 2 ) );
    return 0;
}
Recursive Types (now with Pointers!): Binary Tree

```c
struct chr_tree{
    struct chr_tree *l, *r;
    char var;
}
```

```ada
type chr_tree;
type chr_tree_ptr is access chr_tree;
type chr_tree is record
    left,right:chr_tree_ptr;
    val:character;
end record;
```
Binary Tree with Explicit Pointers: C
Binary Tree in Reference Model: Lisp
Lists

• A list is defined recursively as either the empty list or a pair consisting of an object (which may be either a list or an atom) and another (shorter) list.
• Lists are ideally suited to programming in functional and logical languages.
• In Lisp, in fact, a program is a list, and can extend itself at run time by constructing a list and executing it.
• Lists can also be used in imperative programs.
• We’ll see more in the future!
Problems with Explicit Reclamation

- Explicit reclamation of heap objects is problematic.
- The programmer may forget to deallocate some objects.
  - Causing memory leaks.
  - For example, in the previous example, the programmer may forget to include the delete statement.
- References to deallocated objects may not be reset.
  - Creating dangling references.
Dealing with Dangling References

- **Tombstones**: Use an intermediary device
- **Lock and Keys**: Use a universal key.
Tombstones

new(ptr1);

ptr2 := ptr1;

delete(ptr1);
Locks and Keys

```c
new(ptr1);
ptr2 := ptr1;
delete(ptr1);
```
Garbage Collection

- Automatic reclamation of the space used by objects that are no longer useful:
  - Developed for functional languages.
  - Essential in this programming paradigm. Why?
- Getting more and more popular in imperative languages.
  - Java, C#, Python
- Generally slower than manual reclamation, but it eliminates a very frequent programming error.
Garbage Collection: Techniques

- When is an object no longer useful?
- There are several garbage collection techniques that answer this question in a different manner.
  - Reference counting.
  - Mark-and-sweep collection.
Reference Counting

- Each object has an associated reference counter.

- Keeps reference counter up to date, and recursively deallocates objects when the counter is zero.
Reference Counting: Problems

- **Extra overhead** of storing and updating reference counts.
- **Strong Typing required.**
  - Impossible in language like C.
  - It cannot be used for variant records.
- **It doesn’t work with circular data structures.**
  - This is a problem with this definition of useful object as an object with one or more references.
Reference Counting: Circular Data Structures

Diagram showing reference counting in a circular data structure with nodes "Larry", "Moe", and "Curly".
Mark-and-Sweep Collection

- A better definition of useless is one that cannot be reached by following a chain of valid pointers starting from outside the heap.
- Mark-and-Sweep GC applies this definition.
Mark-And-Sweep

- Algorithm:
- Mark every block in the heap as *useless*.
- Starting with all pointers outside the heap, *recursively* explore all linked data structures.
- Add every block that remain marked to the free list.

- Run whenever free space is low.
Mark-and-Sweep Collection: Problems

- Block must begin with an *indication of its size*.
- A *stack of depth proportional to the longest reference chain is required*.
  - Possible to implement *without a stack*!
  - Pointer reversal algorithm.
- **AND!** We are *usually running low when running the GC*. 
Pointer Reversal Algorithm

- Efficient marking must record the trace it passed.
- Temporarily reversing of pointers traversed by mark (child-pointers become ancestor-pointers).
- Restore pointer fields when tracing back.
- Developed independently by Schorr and Waite (1967) and by Deutsch (1973).
Mark-and-Sweep Collection: Pointer Reversal
Mark-and-Sweep Collection: Pointer Reversal
Mark-and-Sweep Collection: Pointer Reversal
Mark-and-Sweep Collection: Pointer Reversal
Mark-and-Sweep Collection: Pointer Reversal
Mark-and-Sweep Collection: Pointer Reversal
Store-and-Copy

- Use to reduce external fragmentation.

- S-C divides the available space in half and allocates everything in that half until its full.
- When that happens, copy each useful block to the other half, clean up the remaining block, and switch the roles of each half.
Store-and-Copy

- The costs of the stop-and-copy algorithm are twofold:
  - The algorithm requires that *all* live objects be copied every time garbage collection is invoked.
    - If an application program has a large memory footprint, the time required to copy all objects can be quite significant.
  - A second cost associated with stop-and-copy is the fact that it requires twice as much memory as the program actually uses.
    - When garbage collection is finished, at least half of the memory space is unused.
- First proposed by Fenichel & Yochelson in 1969.